RUBY METHODS

Ruby methods are very similar to functions in any other programming language. Ruby methods are used to bundle one or more repeatable statements into a single unit.

Method names should begin with a lowercase letter. If you begin a method name with an uppercase letter, Ruby might think that it is a constant and hence can parse the call incorrectly.

Methods should be defined before calling them otherwise Ruby will raise an exception for undefined method invoking.

Syntax:

```
def method_name [( [arg [= default]]...[, * arg [, &expr ]])]
   expr..
end
```

So you can define a simple method as follows:

```
def method_name
   expr..
end
```

You can represent a method that accepts parameters like this:

```
def method_name (var1, var2)
   expr..
end
```

You can set default values for the parameters which will be used if method is called without passing required parameters:

```
def method_name (var1=value1, var2=value2)
   expr..
end
```

Whenever you call the simple method, you write only the method name as follows:

```
method_name
```

However, when you call a method with parameters, you write the method name along with the parameters, such as:

```
method_name 25, 30
```

The most important drawback to using methods with parameters is that you need to remember the number of parameters whenever you call such methods. For example, if a method accepts three parameters and you pass only two, then Ruby displays an error.

Example:

```
#!/usr/bin/ruby

def test(a1="Ruby", a2="Perl")
   puts "The programming language is #{a1}"
   puts "The programming language is #{a2}"
end
test "C", "C++"
```

```
test
```

This will produce following result:

```
The programming language is C
The programming language is C++
The programming language is Ruby
The programming language is Perl
```

Return Values from Methods:

Every method in Ruby returns a value by default. This returned value will be the value of the last statement. For example:

```
def test
   i = 100
   j = 10
   k = 0
end
```

This method, when called, will return the last declared variable k.

Ruby *return* Statement:

The *return* statement in ruby is used to return one or more values from a Ruby Method.

Syntax:

```
return [expr[`,' expr...]]
```

If more than two expressions are given, the array contains these values will be the return value. If no expression given, nil will be the return value.

Example:

```
return

OR

return 12

OR

return 1,2,3
```

Have a look at this example:

```
#!/usr/bin/ruby

def test
   i = 100
   j = 200
   k = 300
return i, j, k
end
var = test
puts var
```

This will produce following result:

```
100
```

```
200
300
```

Variable Number of Parameters:

Suppose that you declare a method that takes two parameters. Whenever you call this method, you need to pass two parameters along with it.

However, Ruby allows you to declare methods that work with a variable number of parameters. Let us examine a sample of this:

```
#!/usr/bin/ruby

def sample (*test)
   puts "The number of parameters is #{test.length}"
   for i in 0...test.length
      puts "The parameters are #{test[i]}"
   end
end
sample "Zara", "6", "F"
sample "Mac", "36", "M", "MCA"
```

In this code, you have declared a method sample that accepts one parameter test. However, this parameter is a variable parameter. This means that this parameter can take in any number of variables. So above code will produce following result:

```
The number of parameters is 3
The parameters are Zara
The parameters are 6
The parameters are F
The number of parameters is 4
The parameters are Mac
The parameters are 36
The parameters are M
The parameters are MCA
```

Class Methods:

When a method is defined outside of the class definition, the method is marked as *private* by default. On the other hand, the methods defined in the class definition are marked as public by default. The default visibility and the *private* mark of the methods can be changed by *public* or *private* of the Module.

Whenever you want to access a method of a class, you first need to instantiate the class. Then, using the object, you can access any member of the class.

Ruby gives you a way to access a method without instantiating a class. Let us see how a class method is declared and accessed:

```
class Accounts
   def reading_charge
   end
   def Accounts.return_date
   end
end
```

See how the method return_date is declared. It is declared with the class name followed by a period, which is followed by the name of the method. You can access this class method directly as follows:

```
Accounts.return_date
```

To access this method, you need not create objects of the class Accounts.

Ruby *alias* Statement:

This gives alias to methods or global variables. Aliases can not be defined within the method body. The aliase of the method keep the current definition of the method, even when methods are overridden.

Making aliases for the numbered global variables ($1, $2,...) is prohibited. Overriding the builtin global variables may cause serious problems.

Synatx:

```
alias method-name method-name
alias global-variable-name global-variable-name
```

Example:

```
alias foo bar
alias $MATCH $&
```

Here we have defined boo alias for bar and $MATCH is an alias for $&

Ruby *undef* Statement:

This cancels the method definition. An *undef* can not appear in the method body.

By using *undef* and *alias*, the interface of the class can be modified independently from the superclass, but notice it may be broke programs by the internal method call to self.

Synatx:

```
undef method-name
```

Example:

To undefine a method called *bar* do the following:

```
undef bar
```